832204116-ZhangLinyihan

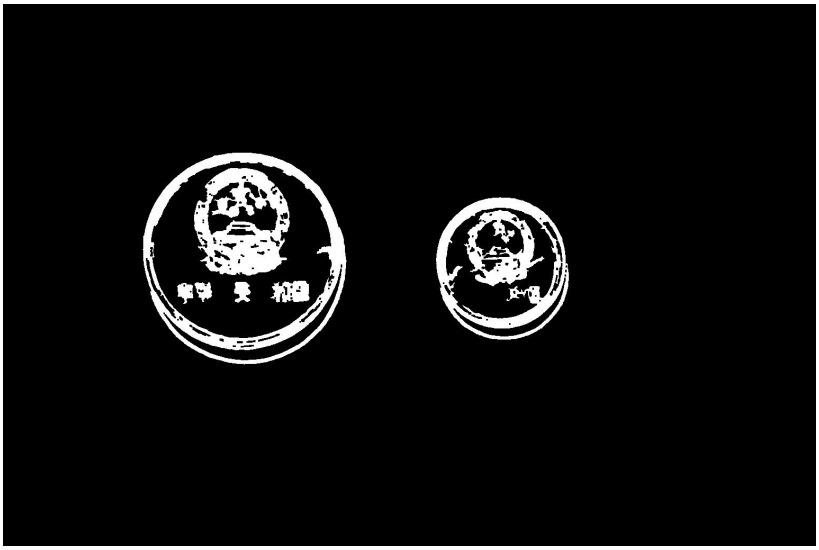
Assignment 1

Hough transform for detecting coins.

Raw image:

Algorithm process:

1. At first, I got the image. Because my kernel is 3*3, so I padded my image (H+2)*(W*2). the I used median filter, Gaussian filter and connected component filter to smooth and clear the texture of the background. Then I used 2 kernel to get the vertical and horizontal gradients via Sobel operator for edge detection.Following picture is the edge detection picture:

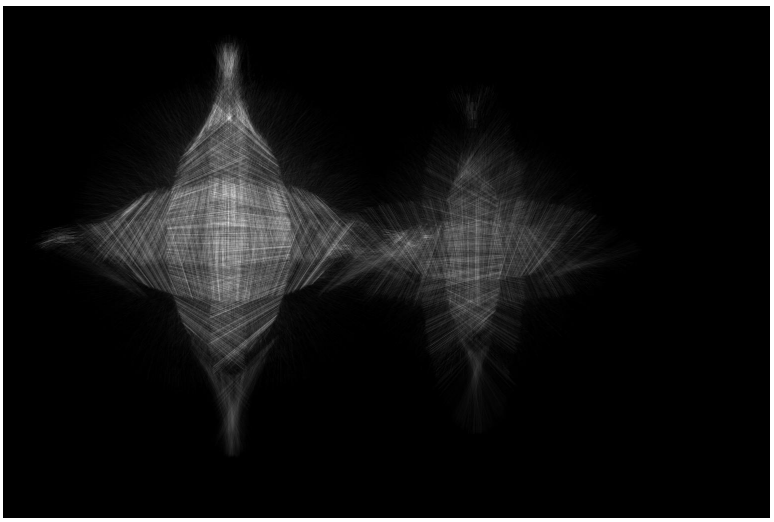


```
def TransformKernel(kernel):
    transform_kernel = kernel.copy()
    for i in range(kernel.shape[0]):
```

```
        for j in range(kernel.shape[1]):
            transform_kernel[i][j] =
kernel[kernel.shape[0]-i-1][kernel.shape[1]-j-1]
    return transform_kernel

def GetPaddedImage(image):
    imagePadded = np.asarray([[0 for x in range(0, image.shape[1]+2)]
for y in range(0, image.shape[0]+2)], dtype=np.uint8)
    imagePadded[1:(imagePadded.shape[0]-1),
1:(imagePadded.shape[1]-1)] = image
    return imagePadded

def Convolution(image, kernel):
    kernel = TransformKernel(kernel)
    imagePadded = GetPaddedImage(image)
    imageConvolution = np.zeros(image.shape, dtype=np.float32)
    for i in range(1, imagePadded.shape[0]-1):
        for j in range(1, imagePadded.shape[1]-1):
            s = 0.0
            for m in range(kernel.shape[0]):
                for n in range(kernel.shape[1]):
                    s += kernel[m][n] * imagePadded[i+m-1][j+n-1]
            imageConvolution[i-1][j-1] = abs(s)
    mx = float(imageConvolution.max())
    if mx > 1e-8:
        imageConvolution /= mx
    return imageConvolution

def PerformSobel(image):
    img = image.copy()
    if img.ndim == 3:
        img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    img = cv2.bilateralFilter(img, d=9, sigmaColor=60, sigmaSpace=60)
    img = cv2.medianBlur(img, ksize=5)
    img = cv2.GaussianBlur(img, (3, 3), 0)

    kernelY = np.asarray([[ 1, 2, 1],
                          [ 0, 0, 0],
                          [-1,-2,-1]])
    kernelX = np.asarray([[-1, 0, 1],
                          [-2, 0, 2],
                          [-1, 0, 1]])
    gradientY = cv2.filter2D(img, cv2.CV_32F, kernelY)
    gradientX = cv2.filter2D(img, cv2.CV_32F, kernelX)
```

```
    theta = np.arctan2(gradientY, gradientX)
    mag   = np.sqrt(gradientX**2 + gradientY**2)
    m = float(mag.max())
    if m > 1e-8:
        mag = mag / m
    return mag, theta
```

2. After getting edge picture of the image, I used NMS to reduce the number of the edge of the coins in the picture and defined a function accumulator to collect votes for potential circle centers using the Hough transform technique. For each edge, I utilized its gradient direction to determine possible centers at multiple radius. The voting process considered both the gradient direction and its opposite dierection to account for circle edge orientation. The accumulator space was structured as a 3D array(X, Y, radius) where each cell represents the voting score for a specific center position and radius.



```
def HoughCircles(image, edge_mask, expect_brighter_inside=True, r_scale=(0.065, 0.20)):
    rows, cols = image.shape[:2]
    mag, theta_map = PerformSobel(image)

    r_min = max(10, int(r_scale[0] * min(rows, cols)))
    r_max = max(r_min+1, int(r_scale[1] * min(rows, cols)))
    radius = list(range(r_min, r_max+1))
    R = len(radius)

    accumulator = np.zeros((rows, cols, R), dtype=np.float32)
    edge01 = (edge_mask > 0).astype(np.uint8)
    ys, xs = np.nonzero(edge01)
    img_gray = image if image.ndim == 2 else cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
```

```
    for idx_r, rr in enumerate(radius):
        for y, x in zip(ys, xs):
            base = float(theta_map[y, x])
            st, ct = math.sin(base), math.cos(base)
            w = float(mag[y, x]) + 1e-6

            a1f = y + (rr * st); b1f = x + (rr * ct)
            if 0.0 <= a1f < rows-1 and 0.0 <= b1f < cols-1:
                if _polarity_vote_allowed(img_gray, y, x, st, ct, rr, +1,
expect_brighter_inside):
                    bilinear_splat(accumulator, a1f, b1f, idx_r, w)

            a2f = y - (rr * st); b2f = x - (rr * ct)
            if 0.0 <= a2f < rows-1 and 0.0 <= b2f < cols-1:
                if _polarity_vote_allowed(img_gray, y, x, st, ct, rr, -1,
expect_brighter_inside):
                    bilinear_splat(accumulator, a2f, b2f, idx_r, w)

    return accumulator, radius
```

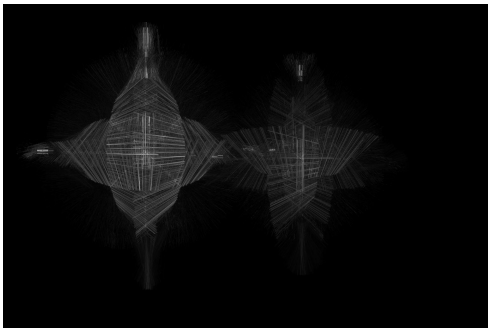
3. After get the radius and center positions, I found a coin included many detection circles. So I add a center merging mechanism and minimum center distance constraint to avoid the problem. The merging process combines overlapping circles by replacing lower-scoring duplicates with higher-scoring ones when their centers are too close. Additionally, a final pruning step ensures that all detected circles maintain a minimum center distance, preventing multiple detections on the same coin while preserving genuine adjacent coins.

```
def FilterCircles(accumulator, radius, t_rel=0.30, min_abs=2,
                  dr_frac=0.50,
                  keep_top=8,
                  tol_frac=0.055, cov_thr=0.20,
                  center_min_dist=28,
                  center_merge_frac=0.80,
                  debug=False):
    H, W, R = accumulator.shape
    cand = []
    for r_idx, r in enumerate(radius):
```

```
        sl = accumulator[:, :, r_idx]
        m = float(sl.max())
        if m <= 0:
            continue
        t = max(t_rel * m, min_abs)
        lmax = local_max_mask(sl)
        ys, xs = np.where((sl >= t) & (lmax > 0))
        for y, x in zip(ys, xs):
            cand.append((float(sl[y, x]), int(y), int(x), int(r_idx)))

    cand.sort(reverse=True, key=lambda z: z[0])

    circles = []
    circle_scores = []
    taken = np.zeros_like(accumulator, dtype=np.uint8)

    for score, y, x, r_idx in cand:
        if taken[y, x, r_idx]:
            continue

        yc_f, xc_f, rc_f = refine_peak(accumulator, radius, y, x, r_idx)

        cov = circle_support_ratio(yc_f, xc_f, rc_f, n_samples=180, tol=max(2.5,
tol_frac * rc_f))
        if cov < cov_thr:
            continue

        merged = False
        for k, (py, px, pr) in enumerate(circles):
            d = ((yc_f - py)**2 + (xc_f - px)**2) ** 0.5
            if d < center_merge_frac * max(pr, rc_f):
                if score > circle_scores[k]:
                    circles[k] = (yc_f, xc_f, rc_f)
                    circle_scores[k] = score
                merged = True
                break
        if merged:
            continue

        circles.append((yc_f, xc_f, rc_f))
        circle_scores.append(score)

        yy = int(round(yc_f)); xx = int(round(xc_f)); rr_pix = int(round(rc_f))
        y0 = max(0, yy - int(0.60 * rr_pix)); y1 = min(H, yy + int(0.60 * rr_pix) + 1)
```

```
        x0 = max(0, xx - int(0.60 * rr_pix)); x1 = min(W, xx + int(0.60 * rr_pix) + 1)
        r0 = max(0, r_idx - max(1, int(dr_frac * rr_pix)))
        r1 = min(R, r_idx + max(1, int(dr_frac * rr_pix)) + 1)
        taken[y0:y1, x0:x1, r0:r1] = 1

        if len(circles) >= keep_top:
            break


    pruned = []
    for cy, cx, r in circles:
        ok = True
        for py, px, pr in pruned:
            if ((cy - py)**2 + (cx - px)**2) ** 0.5 < max(center_min_dist, 0.30 * pr):
                ok = False
                break
        if ok:
            pruned.append((cy, cx, r))

    return pruned


def DetectCircles(img_gray, edges, r_scale=(0.065, 0.20), expect_brighter_inside=True,
**filter_kwargs):
    global _EDGE01_FOR_VERIFY, _DIST_FOR_VERIFY

    edge01 = (edges > 0).astype(np.uint8)
    inv = (1 - edge01) * 255
    _EDGE01_FOR_VERIFY = edge01
    _DIST_FOR_VERIFY = cv2.distanceTransform(inv, cv2.DIST_L2, 3)

    accumulator, radius = HoughCircles(img_gray, edges, expect_brighter_inside,
r_scale=r_scale)

    save_hough_gray(accumulator.max(axis=2), 'hough_space_max.png')
    save_hough_gray(accumulator.sum(axis=2), 'hough_space_sum.png')

    circles = FilterCircles(accumulator, radius, **filter_kwargs)
    return circles
```

4. At last, visualizing the result.



```
if __name__ == "__main__":
    img = cv2.imread('./coins.png', 0)

    mag, theta = PerformSobel(img)
    nz  = mag[mag > 0]
    thr = np.quantile(nz, 0.93) if nz.size else 1.0
    edges01 = (mag >= thr).astype(np.uint8)
    edges = remove_small_components(edges01, min_size=600, connectivity=8)
    edges = cv2.morphologyEx(edges, cv2.MORPH_CLOSE, np.ones((3,3), np.uint8),
iterations=1)
    cv2.imwrite('edges.png', edges)

    found = DetectCircles(
        img, edges,
        r_scale=(0.060, 0.22),
        t_rel=0.33, min_abs=3,
        tol_frac=0.040, cov_thr=0.26,
        keep_top=4, dr_frac=0.28,
        center_min_dist=40,
        debug=True
    )

    result = cv2.cvtColor(img, cv2.COLOR_GRAY2BGR)
    if found:
        print(f"[INFO] Found {len(found)} circle(s):")
        for (cy, cx, r) in found:
            print(f"  center=({cx:.2f},{cy:.2f}), radius={r:.2f}")
            cv2.circle(result, (int(round(cx)), int(round(cy))), int(round(r)),
(0, 0, 255), 2)
            cv2.circle(result, (int(round(cx)), int(round(cy))), 2, (0, 255, 0),
-1)
    else:
        print("[WARN] No circles.")
    cv2.imwrite('houghCoin.png', result)
```